State Design Pattern

Purpose and Motivation

The purpose of the state pattern is to implement a program's functionality as a series of states, each having their own response to inputs. The main idea is to allow the object to change its behaviour without changing its class.

There are two main requirements for a state-based design. First, an object should change its behaviour when its internal state changes. Second, state-specific behaviour should be defined independently. That is, adding new states should not affect the behaviour of existing states.

Intended Use Cases

State is a useful design pattern to implement when an application is required to dynamically change its state and the way that it responds to input in response to prior input. It is most easily implementable when the various expected behaviours of an application can be split up and defined as distinct states.

Vocabulary

Context: A single interface to the outside world, has references to the Concrete State object, which are used to define the current state.

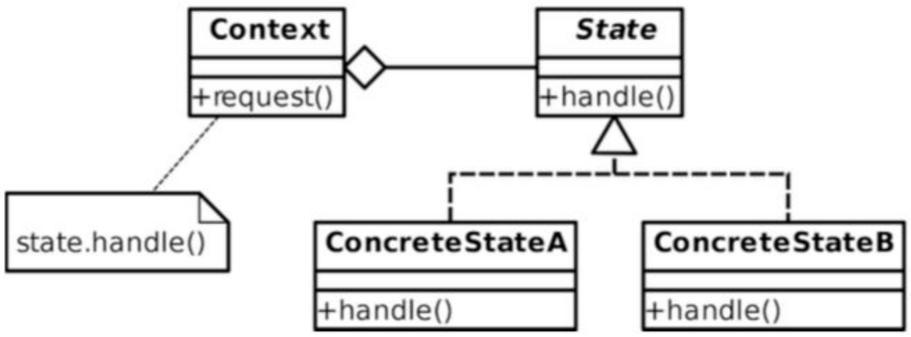
Abstract State: This abstract class is the base class for all concrete state classes. State defines the interface that will be used by the Context object to access the changeable functionality. No state, in terms of fields or properties, is defined in the StateBase class or its subclasses.

Concrete State: The concrete state classes provide the actual functionality that will be used by the Context. Each state class provides behaviour that is applicable to a single state of the Context object. They may also include instructions that cause the Context to change its state.

Structure and Runtime Behaviour

In state-based design, the program interacts with a "context object", which is the object which contains state. All implementation logic of the object is stored in state objects. The context object is in one state at any time, and will use the implementations in its current concrete state object to perform tasks.

The following UML diagram describes a general implementation of the state design pattern, with context object, abstract state interface, and concrete state implementations.



Known Consequences

State-based design reduces coupling in the program which implements it. A given concrete state object will handle its own implementation of the state methods required. The behaviour of the context object will be based on the state that it is currently in. However, some coupling is inevitable as each concrete state class has some knowledge of its sibling's, since they all must implement the same methods.

Positive Consequences

- Supports polymorphism: Easy to add states to support additional behaviour.
- Dynamic behaviour: Allows an object to change its behaviour at runtime depending on its internal state.
- Improves cohesion: State-specific behaviours are aggregated in the ConcreteState classes, placed in one location in the code.

Negative Consequences

- Large number of classes: In a scenario with a lot of states, each state has to be implemented as a concrete state class, therefore requiring more code to be written than perhaps necessary (especially if each state has very little functionality).
- Difficult to debug: Since the program is dynamic and can be in any state at any time given prior input, it is difficult to debug the program at runtime (without knowing its full history of input up until the current time).

NFPs

Improved NFPs

- Complexity: Supports separation of concerns and high cohesion, as each logic is divided to fall under a specific state.

- Scalability: Supports easily adding additional states.
- Dependability: since the states are implemented separately, when one state object introduces failure, the other ones are unaffected.
- Adaptability: it is easy to change existing states or add new ones in order to augment or alter object behaviour.

Inhibited NFPs

- Maintainability: could result in a large number of concrete state classes and more code, making it more difficult to maintain.

Credits

Students:

BadgerCubed (Jason Dias, Spyros Loukidelis, Yiran Sun, Xin Wang)
Team HotSauce (Justin Lim, Tirath Patel, Lucy Yu)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal